ANALYZING THE IMPACT OF DVFS AND SYSTEM NOISE ON MERGE SORT BENCHMARKING THROUGH SYSTEM QUIESCING

ARXIV PREPRINT

Mohammad-Moein Arabi
College of Engineering
University of Tehran
moein.arabi@ut.ac.ir

July 17, 2026

ABSTRACT

Accurate performance evaluation of computer programs requires reliable execution time measurements. However, modern computer systems contain various hardware and software mechanisms that introduce variability into timing measurements, making the observed performance differ from the algorithm's actual behavior. In this study, the execution time of the Merge Sort algorithm was measured for different input sizes under two experimental conditions: normal system operation and a quiesced system. The results demonstrate that system noise produces noticeable fluctuations in execution time, while reducing these sources of interference significantly improves the stability and repeatability of the measurements. These findings highlight the importance of controlling system noise when conducting performance benchmarking and demonstrate that system quiescing provides a more reliable environment for performance evaluation.

Keywords Performance Engineering · Benchmarking · Dynamic Voltage and Frequency Scaling (DVFS) · Merge Sort · System Quiescing · Thermal Throttling · Performance Measurement

1 Introduction

In this project, the anomalous behavior observed in measuring program execution time on modern computer systems is investigated. The primary focus of the experiment is to measure the execution time of the Merge Sort algorithm and analyze the factors that introduce variability into the measured results. One of the most significant factors is Dynamic Voltage and Frequency Scaling (DVFS), which automatically adjusts the processor's operating frequency according to the workload and thermal conditions, thereby affecting program execution time.

As illustrated in Figure 1, execution time may exhibit oscillatory patterns, known as the *Roller Coaster Effect*, instead of following the smooth trend predicted by the algorithm's theoretical time complexity. In addition to DVFS, factors such as background processes, operating system interrupts, processor temperature variations, and other hardware and software characteristics also influence the measurement results.

In this project, besides demonstrating this phenomenon, the *Quiescing System* approach was employed to reduce the impact of these interfering factors. The objective of this approach is to establish a stable execution environment and obtain accurate, reliable, and repeatable measurements so that the true behavior of the algorithm can be evaluated without the influence of system noise.

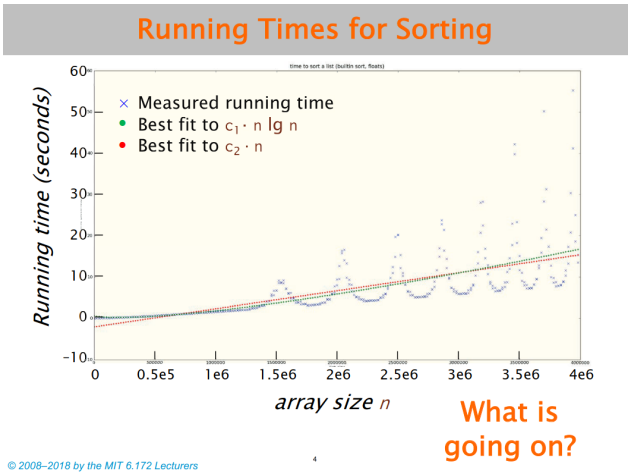

Figure 1: This figure illustrates the *Roller Coaster Effect* in execution time measurements, as presented in the Performance Engineering course at MIT. [Leiserson, 2018, MIT OpenCourseWare, 2019]

2 Background

2.1 Producing Wrong Data Without Doing Anything Obviously Wrong

This paper [Mytkowicz et al., 2009], is a foundational reference in performance engineering. It demonstrates that performance measurements can be significantly biased by "seemingly irrelevant" factors that shift memory alignment. For example:

- Linker Order: Changing the order of .o files in a linker command can have a larger impact on performance than moving from -O2 to -O3 optimization levels.
- Program Name Length: Because an executable's name is stored in an environment variable on the call stack, simply lengthening a program's name can shift stack alignment and slow down data access by causing it to cross page boundaries.

2.2 Quiescing a System

Quiescing a system refers to a set of procedures performed to minimize sources of system noise, thereby enabling more accurate, stable, repeatable, and deterministic execution time measurements. Various factors, including operating system activity, background processes, and processor power management mechanisms, introduce fluctuations in execution time. Therefore, before conducting performance evaluation experiments, these sources of variability should be minimized or controlled as much as possible. As illustrated in Figure 2, the quiescing process systematically reduces external interference, stabilizes hardware behavior, and isolates the benchmarked application, resulting in significantly lower measurement variance and more reliable performance data.

2.3 Major Sources of System Noise

Modern computer systems contain numerous factors that affect the accuracy of execution time measurements. The most important of these include [Leiserson, 2018]:

- Operating system activity: Background daemon processes, scheduled tasks (Cron jobs), interrupts generated by network traffic, and even mouse movements or keyboard input can interfere with timing measurements.
- Hardware features: Dynamic Voltage and Frequency Scaling (DVFS), which dynamically adjusts the processor frequency according to temperature and workload, as well as Turbo Boost, which changes the processor frequency depending on the number of active cores, introduce variability into program execution time.
- Processor architectural features: Hyper-Threading, which allows multiple threads to share processor execution resources, is another factor that contributes to variability in timing measurements.

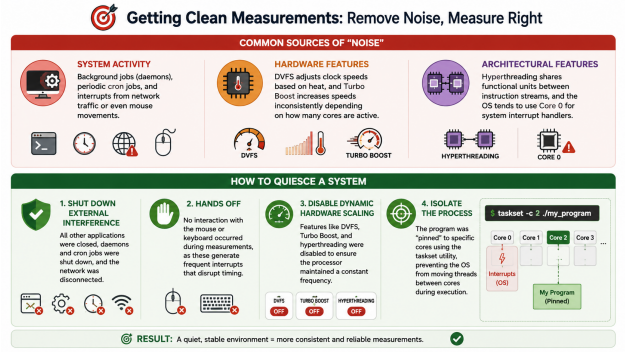

Figure 2: Schematic illustration of the system quiescing process and the reduction of measurement result variance. *Generated by ChatGPT [OpenAI, 2026]*

Table 1: Hardware and software configuration of the experimental platform.

Property	Value
Processor	Intel® Core™ i7-6700HQ
Architecture	x86_64
Physical Cores	4
Logical CPUs	8
Base Frequency	2.60 GHz
Maximum Frequency	3.50 GHz
L3 Cache	6 MiB
Memory	12 GiB RAM
Operating System	Ubuntu 25.10
Kernel	Linux 6.17.0-14-generic
Compiler	GCC 15.2.0
Compiler Flags	`-O2`

3 Experimental Methodology

3.1 Experimental Platform

All experiments were conducted on the hardware and software platform summarized in Table 1. The benchmark program was compiled using GCC version 15.2.0 with the `-O2` optimization level. Unless otherwise stated, all reported results were obtained on this platform.

3.2 Code Description

To conduct the experiment, a program was developed in C to measure the performance of the Merge Sort algorithm for arrays of different sizes. The program first creates and initializes an array of a specified size, then measures the execution time of the sorting algorithm while recording the CPU temperature. This procedure is repeated for different array sizes, and all collected measurements are finally stored in a CSV file.

3.2.1 Array Size Selection

The maximum array size was set to 6,000,000 elements. In the reference [Leiserson, 2018], the maximum size was 4,000,000 elements; however, in this project, the limit was increased to better observe the algorithm's behavior for larger input sizes.

```
int max = 6 * 1000 * 1000;
int min = 1;
int step = 20 * 1000;
```

Table 2: Sample of the data stored in the CSV file

Array Size	Delay (sec)	CPU Temperature (°C)
1	0.000000	49.00
20001	0.002278	49.00
40001	0.005453	49.00
60001	0.009425	49.00
80001	0.015302	49.00
100001	0.017824	49.00
120001	0.017458	49.00
140001	0.021326	49.00
160001	0.023287	49.00

3.2.2 Measuring CPU Temperature and Execution Time

Before executing the algorithm, the CPU temperature is recorded to monitor temperature variations throughout the experiment. Under Linux, the CPU temperature is obtained from the following file:

```
cat /sys/class/thermal/thermal_zone0/temp
```

The execution time of the algorithm is then measured using the `clock_gettime()` function with the `CLOCK_MONOTONIC` clock.

```
double current_temp = get_cpu_temperature();

clock_gettime(CLOCK_MONOTONIC, &start);
my_sort(A, n);
clock_gettime(CLOCK_MONOTONIC, &end);
```

3.2.3 Execution Progress and Data Collection

After each measurement, the program displays its execution progress in the terminal, while the array size, execution time, and CPU temperature are stored in memory.

```
printf("progress %.1f%%, size %d, time %.6f sec, temp %.2f C\n",
       progress, n, tdiff, current_temp);

storage[measurement_count].size = n;
storage[measurement_count].delay = tdiff;
storage[measurement_count].temp = current_temp;
```

3.2.4 Saving the Results to a CSV File

After all measurements have been completed, the data stored in memory are written to a CSV file for subsequent visualization and analysis.

```
FILE *csv_file = fopen("measurements.csv", "w");
fprintf(csv_file,
        "Array Size,Delay (sec),CPU Temperature (C)\n");

for (int i = 0; i < measurement_count; i++) {
    fprintf(csv_file, "%d,%.6f,%.2f\n",
            storage[i].size,
            storage[i].delay,
            storage[i].temp);
}
```

A sample of the generated CSV file is shown in Table 2.

3.2.5 Plotting the Results

After completing the measurements, the collected data were stored in a CSV file for subsequent analysis. The data were processed using the Pandas library [Wes McKinney, 2010, pandas development team, 2020], while the execution time measurements were visualized using Matplotlib [Hunter, 2007]. To compare the experimental results with the theoretical time complexity of the Merge Sort algorithm, an $n \log n$ curve was fitted to the measured data using SciPy [Virtanen et al., 2020], allowing the observed behavior of the algorithm to be compared with its theoretical model.

3.3 Experimental Procedure

The first experiment was conducted without applying the system quiescing procedure. The benchmark program was compiled using the GCC compiler with the `-O2` optimization level:

```
gcc -O2 mergesort_benchmark.c -o mergesort_benchmark
```

After executing the program, the execution time of the algorithm was measured for different array sizes, and the results were stored in a CSV file.

The experiment was then repeated after applying system quiescing. Several steps were performed under Linux to reduce measurement noise, as described below.

3.3.1 Steps Taken to Quiesce the System

To reduce system noise and improve the repeatability of the experiments, the following measures were taken:

- Reducing system noise: All unnecessary applications were closed, and the wireless network connection (Wi-Fi) was disabled.
- Avoiding user interaction: No keyboard or mouse interaction occurred during the experiments, since such interactions generate interrupts that increase measurement error.
- Disabling dynamic processor features: Features such as DVFS, Turbo Boost, and Hyper-Threading were disabled to ensure that the processor operated at a fixed frequency throughout the experiments.
- Pinning the program to a dedicated CPU core: To prevent process migration between processor cores and to avoid the additional interrupts typically associated with core 0, the program was pinned to a dedicated CPU core using the Linux `taskset` command.

Setting the CPU Governor

```
for cpu in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
do
    echo performance | sudo tee "$cpu"
done
```

Disabling Intel Turbo Boost

```
echo 1 | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo
```

Disabling Network Connectivity

```
nmcli networking off
```

Disabling Hyper-Threading

```
echo off | sudo tee /sys/devices/system/cpu/smt/control
```

Pinning the Program to a CPU Core

```
taskset -c 2 ./mergesort_benchmark
```

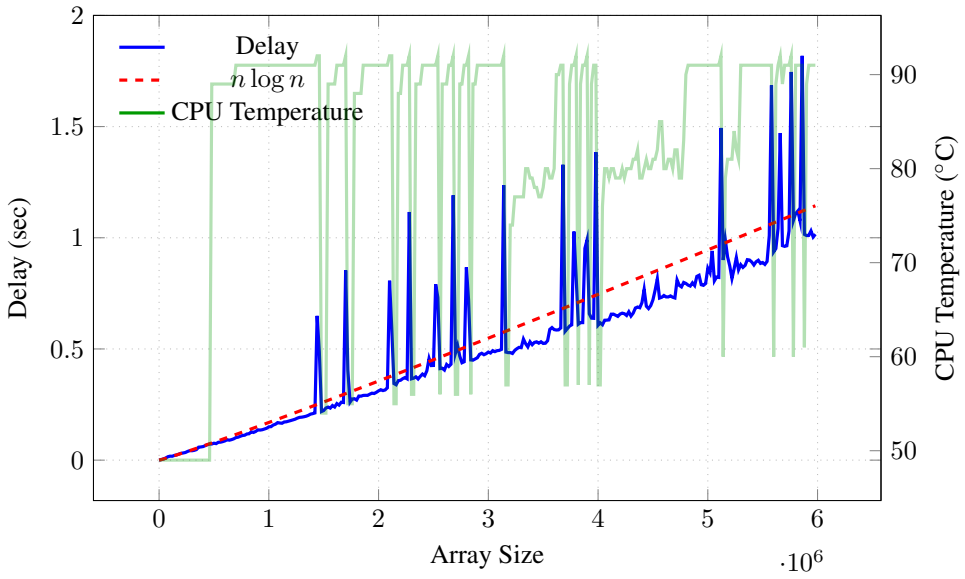

Figure 3: Execution time measurements obtained before applying system quiescing.

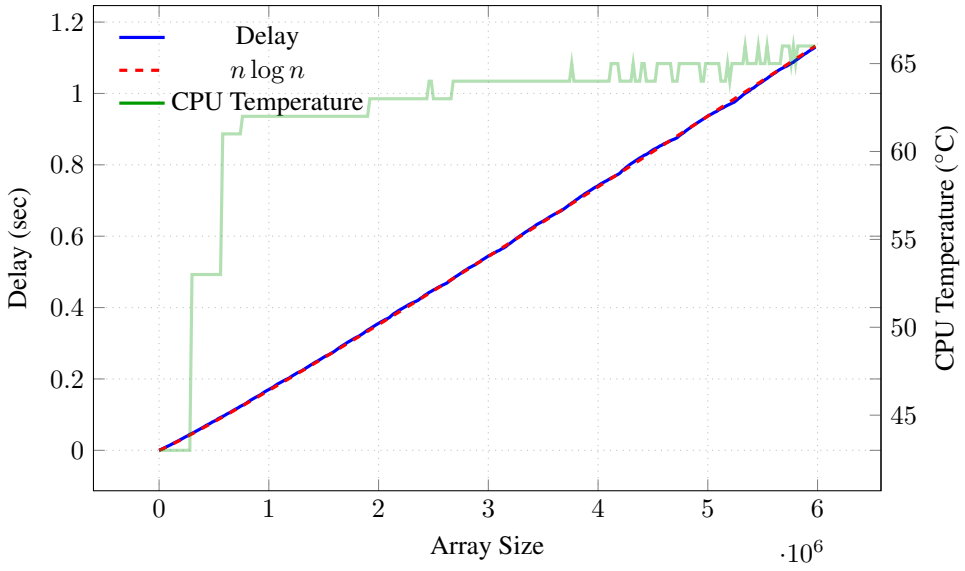

Figure 4: Execution time measurements obtained after applying system quiescing.

4 Results

To evaluate the impact of system noise on execution time measurements, two sets of experiments were performed. In the first experiment, measurements were collected under normal operating conditions without applying system quiescing. In the second experiment, the same measurements were repeated after applying the quiescing procedure.

Figure 3 shows the execution time measurements obtained under normal system conditions.

Figure 4 presents the execution time measurements obtained after applying the system quiescing procedure.

5 Discussion and Conclusion

A comparison of the two experiments shows that, before system quiescing, the execution time exhibited significant fluctuations, and the resulting curve deviated noticeably from a smooth trend. These fluctuations are primarily caused by factors such as dynamic CPU frequency scaling, background operating system activity, and other sources of system noise.

After applying the quiescing procedure, the variability of the measurements was substantially reduced, resulting in a smoother execution time curve. The CPU temperature also remained more stable throughout the experiments, and the measured execution times more closely followed the theoretical $n \log n$ complexity curve. These results demonstrate that minimizing external sources of interference leads to a more deterministic execution environment, thereby improving the accuracy, repeatability, and reliability of performance measurements.

Code and Data Availability

The complete source code, experimental scripts, benchmark data, and instructions for reproducing the results presented in this paper are publicly available at:

`https://github.com/ILoveBacteria/DVFS-performance-anomalies`

Acknowledgment

Generative AI tools [OpenAI, 2026] were used to assist with language refinement and code development. The authors reviewed and verified all generated content.

References

Charles E. Leiserson. Measurement and timing. MIT OpenCourseWare, 6.172 Performance Engineering of Software Systems, Lecture 10, 2018. URL `https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/4d7f4bb31bf1ed90c669a11867d36d36_MIT6_172F18_lec10.pdf`. Accessed: 2026-07-17.

MIT OpenCourseWare. 10. Measurement and Timing, September 2019. URL `https://www.youtube.com/watch?v=LvX3g45ynu8`.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 265–276. ACM, 2009. doi:10.1145/1508244.1508275. URL `https://doi.org/10.1145/1508244.1508275`.

OpenAI. Chatgpt (gpt-5.5). `https://chatgpt.com/`, 2026. Large language model. Accessed: 2026-07-17.

Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi:10.25080/Majora-92bf1922-00a.

The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL `https://doi.org/10.5281/zenodo.3509134`.

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.